ICS 33.160.25

CCS M74

UHD World Association
世界超高清视频产业联盟

世 界 超 高 清 视 频 产 业 联 盟 标 准

T/UWA XXXX-XX-2022

超高清视频处理算法接口技术要求

Technical requirements of ultra-high definition system for teleconsultation

（征求意见稿）

在提交反馈意见时，请将您知道的相关专利连同支持性文件一并附上

XXXX－XX－XX 发布　　XXXX－XX－XX 实施

世界超高清视频产业联盟

目　　次

前　　言

本文件按照GB/T 1.1-2020《标准化工作导则 第1部分:标准化文件的结构和起草规则》的规定起草。
请注意本文件的某些内容可能涉及专利。本文件的发布机构不承担识别专利的责任。
本文件由世界超高清视频产业联盟（UWA）提出并归口。
本文件起草单位：。
本文件主要起草人：。

引　言

本文件的发布机构提请注意，声明符合本文件时，可能使用以下涉及的相关专利：

——一种数据、视频处理方法及装置（专利申请号202211528132.2）；

——影像处理方法、影像处理装置、存储介质（专利申请号202210343186.5）；

——视频转码方法、系统及电子设备（专利申请号202110336370.2）；

本文件的发布机构对于该专利的真实性、有效性和范围无任何立场。

该专利持有人已向本文件的发布机构承诺，其愿意同任何申请人在合理且无歧视的条款和条件下，就专利授权许可进行谈判。该专利持有人的声明已在本文件的发布机构备案。相关信息可以通过以下联系方式获得：

联系人：苏京

通讯地址：北京市经济技术开发区地泽路9号

邮政编码：100176

电子邮件：sujing@boe.com.cn

电　　话：13811947489

网址：https://www.boe.com.cn/

请注意除上述专利外，本文件的某些内容仍可能涉及专利。本文件的发布机构不承担识别专利的责任。

超高清视频处理算法接口技术要求

1 范围

本文件规定了超高清视频处理算法接口，包括算法任务接口和算法服务接口。

本文件适用于超高清视频处理算法的接入与应用，也可用于指导智能超高清视频处理系统与算法包、算法服务的系统集成与开发术语和定义。

2 规范性引用文件

下列文件对于本标准的应用是必不可少的。凡是注日期的引用文件，注日期的版本适用于本标准。凡是不注日期的引用文件，其最新版本（包括所有的修改单）适用于本标准。

GY/T 299.1-2016 高效音视频编码第一部分：视频

ISO/IEC 14882:2011 C++11标准

3 术语和定义

下列术语和定义适用于本文件。

3.1 算法任务 algorithm task

实现对视频文件进行解码、降噪、超分辨率、帧率提升、动态范围提升、编码等功能的一系列算法库的集合。

3.2 算法部署 algorithm deployment

算法组件及其依托的基础运行环境，在硬件设备上安装与实施的过程。

3.3 算法服务 algorithm service

算法组件、组件运行逻辑、组件依赖及其基础运行环境的完整封装包。

4 缩略语

下列缩略语适用于本文件。

HDR: 高动态范围（High Dynamic Range）

GPU: 图形处理器（Graphics Processing Unit）

AI: 人工智能（Artificial Intelligence）

C++11：C++11 标准（ISO/IEC 14882:2011 Information technology Programming languages）

5 应用环境和接口分类

5.1 应用环境

超高清视频处理算法接口是一种可用于视频超高清处理的通用算法接口，算法采用多线程充分调度硬件资源，实现视频处理算法任务高并发处理，涉及的内容包括算法服务接口和算法任务接口。

算法接口的应用环境见图1。

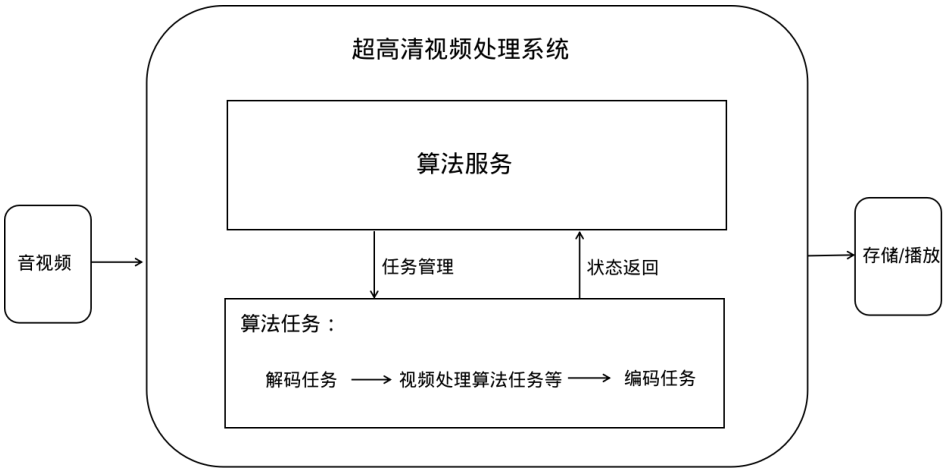

图 1 超高清视频处理算法接口应用环境示意图

应用环境的功能如下：

a) 算法服务是对算法任务的流程化管理，对算法任务进行高并发调度，实现硬件资源最大化利用，以达到最高的处理效率。具体功能包括任务创建、任务调度、资源管理、数据同步，监控算法任务的返回状态并作出响应。

b) 算法任务是对音视频进行算法处理的所有任务的集合，包含但不限于解码任务、视频处理算法任务、编码任务；其中视频处理算法任务包括AI算法任务和其他非AI传统算法任务，AI算法任务包含但不限于视频降噪、超分辨率，帧率提升、HDR等超高清视频处理算法功能。其中解码任务是对输入的视频文件进行解码，得到视频数据的媒体数据（包含图像帧数据和音频数据）。AI算法任务包含但不限于视频降噪、超分辨率，帧率提升、HDR等超高清视频处理算法功能。编码任务是对经过算法任务处理完成的媒体数据进行编码，输出视频文件。

c) 算法任务间传递的数据包括媒体数据、元数据等。元数据用于记录超高清视频处理名称、超高清视频处理版本、厂家名称、处理算法名称、处理算法参数、编解码参数等信息。

d) 经过超高清视频处理系统输出的视频文件可存储在本地硬盘，或者在终端显示设备上进行渲染播放。

5.2 接口分类

按照算法功能和实现方式进行分类，具体如下：

a) 算法服务接口，用于对算法任务的流程化管理。

b) 算法任务接口，用于调用具体的算法功能，包括但不限于解码任务，视频处理算法任务，编码任务等。

5.3 接口要求

a) 接口函数以动态链接库的形式发布，开放给用户使用。

b) 适用于 Winsdows 系统、Linux 系统、Centos 系统，对于不同的操作系统，编译成不同的动态链接库。

c) 内存管理在具体的算法任务中实现。

d) 接口函数基于 C++标准程序库进行开发。

e) 附录 A 为函数原型说明和 c++语言调用示例。

f) 附录 B 为函数返回错误码详细说明。

g) 附录 C 为依赖库版本详细说明。

6 算法接口

包含算法服务和算法任务的配置流程和接口函数详情。

6.1 算法服务

算法服务在超高清处理系统中负责管理算法任务和调度资源，包括三部分：任务创建类、数据同步类、流程管理类。

6.1.1 算法服务接口概述

图2为算法服务接口环境示意图

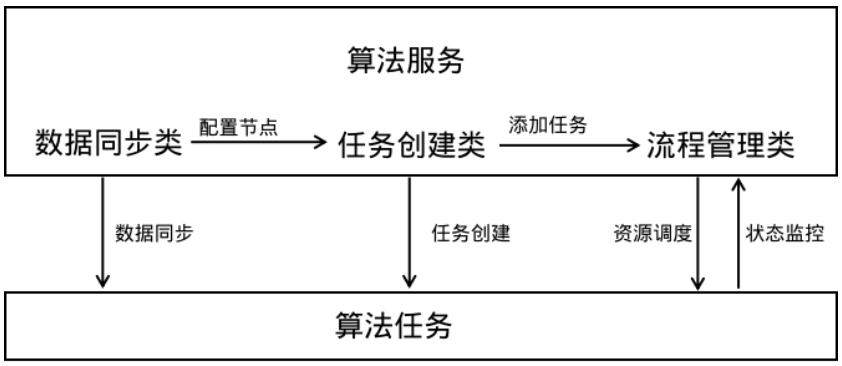

图 2 视频处理算法服务接口环境示意图

算法服务包含任务创建类、数据同步类、流程管理类，这三个类实现了对算法任务的流程化管理。任务创建类定义了算法任务实现的预定义规则，以及配置算法任务数据同步节点和从节点中获取数据的方法。数据同步类实现了算法任务的数据同步传递机制，以保障不同算法任务间数据的同步传递。

流程管理类实现了资源管理、任务调度、状态监控，其中资源管理包含算法任务资源分配和释放；任务调度包含添加算法任务和删除算法任务；状态监控可以接受算法任务的返回信息，判断算法任务的运行状态，包括任务运行中，任务出现错误，任务结束。

在超高清处理系统中，通过算法服务对算法任务的管理，能够实现图像帧序列按顺序经过各个算法任务进行数据处理，每个算法任务只专注自身的处理步骤，判断自身的资源占用情况，只要当前任务资源空闲，就可以接受相邻的上一任务的数据，并激活该任务进行数据处理，也就是说各个算法任务可以并行处理。

6.1.2 算法服务接口

算法服务接口应符合表 1 的规定，若无特殊说明，接口函数的状态返回值均为：调用成功，返回 0，调用失败，返回错误码，详细错误码类型见附录 B。

表 1 算法服务接口列表

名称	主要成员	说明
数据同步类	构造函数	见6.1.2.1
	释放资源函数	见6.1.2.1
	可写信号量	见6.1.2.1
	可读信号量	见6.1.2.1
	数据节点类	见6.1.2.1
流程管理类	初始化函数	见6.1.2.2
	添加任务函数	见6.1.2.2
	算法任务线程函数	见6.1.2.2
	启动任务函数	见6.1.2.2
	关闭任务函数	见6.1.2.2
	算法任务存储表	见6.1.2.2
任务创建类	数据处理函数	见6.1.2.3
	资源释放函数	见6.1.2.3

	前向节点配置函数	见6.1.2.3
	后向节点配置函数	见6.1.2.3
	数据等待函数	见6.1.2.3
	数据传递函数	见6.1.2.3
	前向节点	见6.1.2.3
	后向节点	见6.1.2.3
	前向节点数据	见6.1.2.3
	后向节点数据	见6.1.2.3

6.1.2.1 数据同步类

构造函数，是数据同步类的构造函数。函数无返回值，在实例化类对象的时候会自动调用，函数对数据同步类的成员变量初始化赋值，具体成员变量应符合表 1 的规定，可读信号量和可写信号量确保算法任务间数据传递的同步性。数据同步类通过对数据节点的管理，保障算法任务间的数据传递和数据同步。

释放资源函数，目的是释放数据同步类所占资源。

可写信号量，为数据同步类的成员变量，类型为信号量，该值大于 0 时，表示数据可写，该值小于 0 时，表示数据不可写。

可读信号量，为数据同步类的成员变量，成员变量，类型为信号量，该值大于 0 时，表示数据可读，该值小于 0 时，表示数据不可读。

数据节点类，为数据同步类的成员变量，定义了算法任务数据传递所需要的具体操作，数据节点类主要成员应符合表2的规定。

表 2 数据节点类主要成员说明表

名称	主要类成员列表	说明
数据节点类	数据链表	存储了数据的具体内容，以查找表的形式定义
	初始化函数	根据传入的参数创建数据节点类对象

	获取数据函数	可根据关键字查找对应的数据，函数返回为NULL表示失败，否则为对应数据内存地址
	增加数据函数	可增加数据链表中的数据
	更改数据函数	可根据关键字更改对应的数据，函数返回为NULL表示失败，否则为更改前的数据内存地址
	移除数据函数	可根据关键字移除数据链表中的数据，函数返回表示NULL为失败，否则为移除前的数据内存地址

数据同步类实例化后的对象，通过前向节点配置函数和后向节点配置函数传递给任务创建类，后赋值给任务创建类的成员变量前向节点和后向节点，而任务创建类的另外两个成员变量前向节点数据和后向节点数据则是从前向节点和后向节点中取值，取值的对象是实例化后的数据节点类。

6.1.2.2 流程管理类

流程管理类实现对算法任务的流程化管理，可为算法任务分配并管理资源、调度算法任务，检测算法任务状态。

初始化函数，目的是初始化视频处理流程管理框架类。

添加任务函数，目的是将算法任务添加至视频处理列表中，函数参数应符合表 3 的规定。

表 3 添加任务函数参数列表

函数名称	输入/输出类型	参数说明
添加任务函数	输入参数	指向算法任务的指针，见6.2

算法任务线程函数，目的是为算法任务线程创建提供函数指针，使每个算法任务都在单独的线程中处理，启动任务函数会调用该指针进行线程创建。函数返回值：无。

启动任务函数，目的是创建线程并启动流程管理框架内的全部算法任务。

关闭任务函数，目的是关闭流程管理框架内的全部算法任务，并释放算法任务所占资源。

算法任务存储表，为流程管理类的成员变量，存储了全部算法任务的链表。

6.1.2.3 任务创建类

任务创建类是算法任务创建的预定义规则，算法任务基于此规则创建后，可添加至算法服务中进行流程化管理。

初始化函数，目的是创建算法任务所需的资源，为算法任务创建必须重载实现的函数。

数据处理函数，目的是实现算法任务的具体操作，为算法任务创建必须重载实现的函数。

资源释放函数，目的是释放算法任务的资源，为算法任务创建必须重载实现的函数。

前向节点配置函数，目的是设置算法任务的前向节点。

后向节点配置函数，目的是设置算法任务的后向节点。

前向节点配置函数和后向节点配置函数，其配置的参数类型为数据同步类，目的是确定当前算法任务在处理流程中的位置，并且保证算法任务间数据传递的同步。启动全部算法任务之前，要配置好算法任务的前后节点，其参数列表应符合 4 的规定。

表 4 算法任务节点配置函数参数列表

函数名称	输入/输出类型	参数说明
算法任务前向节点配置函数	输入参数	参数类型为数据同步类，见6.1.2.1
算法任务后向节点配置函数	输入参数	参数类型为数据同步类，见6.1.2.1

数据等待函数，目的是等待上一算法任务的数据，参数类型应符合表 5 的规定。

数据传递函数，目的是将数据传递给下一算法任务，参数类型应符合表 5 的规定。

数据等待函数和数据传递函数，通过修改数据同步类成员变量可读信号量和可写信号量，使算法任务可以在数据节点中获取以及数据传递，其参数列表应符合表 5 的规定。

表 5 数据等待/传递函数参数列表

函数名称	输入/输出类型	参数说明
数据等待函数参数列表	输入参数	参数类型为数据节点类，见表2
数据传递函数参数列表	输入参数	参数类型为数据节点类，见表2

6.1.3 算法服务接口配置流程

算法服务接口配置流程应满足图 3 的规定。

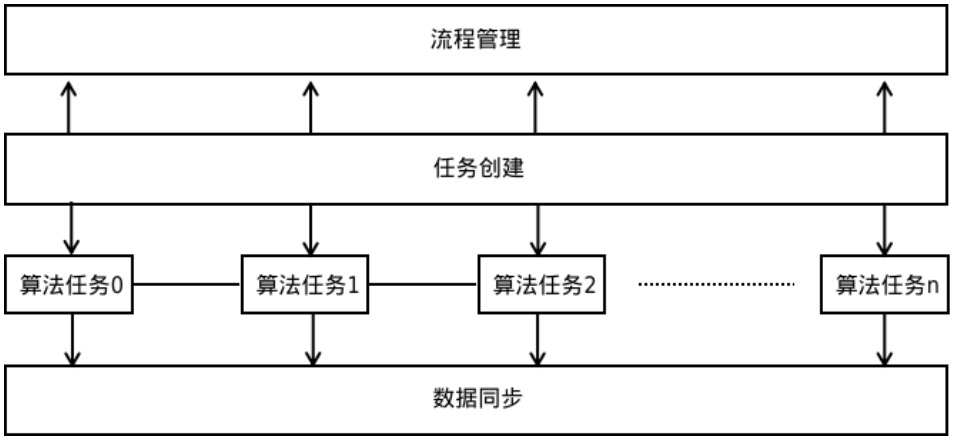

图 3 算法服务接口配置流程

图3为算法服务接口配置流程，通过对任务创建类进行派生的形式，创建具体的算法任务，即图中算法任务0、算法任务1、算法任务2和算法任务n。算法任务创建后，可添加至流程管理类中进行管理，流程管理会为每一个算法任务创建线程，分配运算资源。通过对数据同步类实例化的方式，创建算法任务间传递的对象，并调用配置前向/后向节点函数来进行算法任务节点配置，数据同步类可保障算法任务在并行处理的情况下能够有序传输。全部算法任务添加完成，配置好前后节点，流程管理类可调用启动任务函数来启动全部任务。流程管理类可通过监测算法任务输出节点数据为空后，关闭算法任务并释放资源。

6.2 算法任务

6.2.1 算法任务接口概述

超高清视频处理算法任务接口应用环境见图 4。

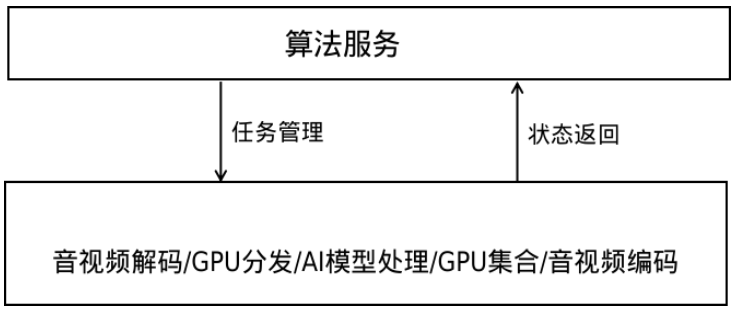

图 4 视频处理算法任务接口环境示意图

算法任务包含音视频解码、GPU分发、AI模型处理、GPU集合、音视频编码处理。AI模型处理需要用到GPU资源，GPU分发任务和GPU集合任务可高效使用GPU资源进行AI模型的加速，算法任务在运行时，只专注于自身的任务模块，包括任务待处理数据的接收、具体算法步骤的实现、输出数据的传递。

音视频解码任务负责音视频的解码，GPU分发任务负责将解码后的图像帧数据按照顺序编号后发送给GPU资源，AI模型处理任务负责对图像帧序列进行AI模型算法处理（包含但不限于视频降噪、超分辨率，帧率提升、HDR等超高清视频处理算法功能），GPU集合任务负责将AI模型处理后的图像帧数据按照编号顺序进行排序，音视频编码任务将排序后的数据编码成视频文件。

6.2.2 算法任务数据类型

6.2.2.1 视频流编码格式类型

用于传递视频流编码格式信息，采用枚举形式。

编码格式类型枚举成员应符合表 6 的规定。

表 6 视频流编码格式类型枚举成员说明表

数据类型	枚举成员列表
视频流编码格式类型	MPEG1VIDEO编码器
	MPEG2VIDEO编码器
	MPEG4编码器
	RAWVIDEO编码器
	WMV1编码器
	WMV2编码器
	H264编码器
	AVS编码器
	VP5编码器
	VP6编码器
	VP8编码器
	HEVC编码器
	H265编码器
	VP7编码器
	AVS2编码器
	可拓展

6.2.2.2 视频流像素格式类型

用于传递视频流像素格式信息，采用枚举形式。

像素格式类型枚举成员说明应符合表 7 的规定。

表 7 视频流像素格式类型枚举成员说明表

数据类型	枚举成员列表
视频流像素格式	YUV420P像素格式
	RGB24像素格式
	BGR24像素格式
	YUV422P像素格式

	YUV444P像素格式
	GRAY8像素格式
	PAL8像素格式
	UYVY422像素格式
	NV12像素格式
	NV21像素格式
	ARGB像素格式
	RGBA像素格式
	ABGR像素格式
	BGRA像素格式
	GRAY16BE像素格式
	GRAY16LE像素格式
	YUV440P像素格式
	YUV420P10像素格式
	YUV422P10像素格式
	YUV444P10像素格式
	P010像素格式
	NV24像素格式
	NV42像素格式
	可扩展

6.2.2.3 视频流色域类型

用于传递视频流色域信息，采用枚举形式。

色域类型枚举成员应符合表 8 的规定。

表 8 色域类型枚举成员说明表

数据类型	枚举成员列表
色域类型	RGB
	BT.709
	FCC
	BT.470 BG
	SMPTE 170 M
	SMPTE 240 M

	YCOCG
	BT.2020 NCL
	BT.2020 CL
	SMPTE 2085
	Chroma-derived NCL
	Chroma-derived CL
	ICtCp
	可扩展

6.2.2.4 HDR 标准类型

用于传递 HDR 标准信息，采用枚举形式。

HDR 标准信息枚举成员应符合表 9 的规定。

表 9 HDR 标准类型枚举成员说明表

数据类型	元数据类型	枚举成员列表
HDR标准类型	静态元数据	PQ HDR10 标准
		HLG HDR 标准
		可拓展
	动态元数据	PQ HDR10+ 标准
		Dolby Visio 标准
		HDRVivid 标准
		可拓展

6.2.2.5 元数据信息枚举

用于传递视频编码元数据信息，采用枚举形式。

元数据信息枚举成员应符合表 10 的规定。

表 10 元数据信息枚举成员说明表

数据类型	枚举成员列表	
元数据信息	超高清视频处理名称	帧率
	超高清视频处理版本	视频编码方式
	厂家名称	色彩空间
	处理算法名称1	位深

	处理参数1	码率
	…	像素格式
	处理算法名称2	音频声道数
	处理参数2	音频采用率
	…	音频编码方式
		HDR标准名称
		时间信息
		…

6.2.2.6 音频流编码格式类型

用于传递音频流编码格式信息，采用枚举形式。

音频编码格式类型枚举成员应符合表 11 的规定。

表 11 音频编码格式类型枚举成员说明表

数据类型	枚举成员列表
音频流编码格式类型	AAC编码器
	PCM编码器
	可扩展

6.2.2.7 视频文件封装类型

用于传递视频文件封装类型信息，采用枚举形式。

视频文件封装类型枚举成员应符合表 12 的规定。

表 12 视频文件封装类型枚举成员说明表

数据类型	枚举成员列表
视频文件封装类型	MP4文件
	MXF文件
	AVI文件
	可扩展

6.2.2.8 算法任务所需的 AI 模型类型

用于传递算法任务所需的 AI 模型信息，采用枚举形式。

算法任务所需的 AI 模型枚举主要成员应符合表 13 的规定。

表 13 算法任务所需的 AI 模型文件枚举主要成员说明表

数据类型	枚举主要成员列表
算法任务所需的AI模型文件	降噪模型
	超分模型
	插帧模型
	HDR模型
	细节修复模型
	可扩展

6.2.2.9 算法任务数据传递配置信息

用于传递算法任务配置信息，采用字符串形式。

算法任务配置信息应符合表 14 的规定。

表 14 算法任务配置信息说明表

数据类型	主要成员列表
算法任务配置信息	输入文件字符串
	算法字符串
	编码器字符串
	像素格式字符串
	可调度GPU资源字符串
	输出文件名字符串
	封装格式字符串
	HDR标准类型字符串

6.2.3 算法任务接口函数

视频处理算法任务接口函数应符合表 15 的规定，表格中的算法任务类均为任务创建类派生的子类。函数返回值若无特殊说明，且不为类的构造函数，接口函数的状态返回值均为：调用成功，返回 0，调用失败，返回错误码，详细错误码类型见附录 B。

表 15 视频处理算法任务接口函数列表

接口类	接口函数	章条号
音视频解码任务类	初始化函数	6.2.3.1
	数据处理函数	6.2.3.1
	释放资源函数	6.2.3.1
	获取视频封装格式信息函数	6.2.3.1

	获取视频帧率函数	6.2.3.1
	获取视频总帧数函数	6.2.3.1
	音视频解码类	6.2.3.1
视频编码任务类	初始化函数	6.2.3.2
	数据处理函数	6.2.3.2
	释放资源函数	6.2.3.2
	音视频编码类	6.2.3.2
音频编码任务类	初始化函数	6.2.3.3
	数据处理函数	6.2.3.3
	资源释放函数	6.2.3.3
	音视频编码类	6.2.3.3
GPU分发任务类	初始化函数	6.2.3.4
	数据处理函数	6.2.3.4
	释放资源函数	6.2.3.4
GPU集合任务类	初始化函数	6.2.3.5
	数据处理函数	6.2.3.5
	释放资源函数	6.2.3.5
AI模型处理任务类	初始化函数	6.2.3.6
	数据处理函数	6.2.3.6
	释放资源函数	6.2.3.6

6.2.3.1 视频解码任务类接口函数

视频解码任务类用于对输入的视频文件或者视频流等进行解码。

初始化函数，目的是为了初始化视频解码任务，将音视频解码类传递给音视频解码任务，函数参数应符合表 16 的规定。

表 16 初始化函数参数列表

参数	输入/输出类型	参数说明
音视频解码类	输入参数	音视频解码类指针

数据处理函数，是视频解码任务类的主体，派生自任务创建类，其实现了从解码类中获取每一帧数据，并把数据传递给其他任务。

资源释放函数，目的是为了释放视频解码任务类所占资源，销毁类成员变量。

获取视频封装格式信息函数，目的是从解码类中获取输入视频的封装格式。

获取视频帧率函数，目的是从解码类中获取输入视频的帧率。

获取视频总帧数函数，目的是从解码类中获取输入视频的总帧数。

音视频解码类实现了视频的解码功能，其封装了获取解码帧数据，及输入视频的相关原始编码信息的函数，类成员应符合表 17 的规定。接口函数的参数详情在附录 A 中。

表 17 音视频解码类主要成员说明表

名称	主要类成员列表	说明
视频解码类	打开视频文件函数	函数根据视频文件路径或视频流来源解析文件信息，创建解码环境
	获取音视频帧数据函数	函数从视频文件中读取一帧，若该帧为视频，则进行解码，将数据传递出去，并返回0;若帧为音频，则直接将音频包传递出去，并返回1;若需要更多帧数据，则返回-5;若视频文件解码完成，则返回-1;若函数返回上述值以为的其他值，表示解码失败。
	获取视频宽函数	返回视频宽
	获取视频高函数	返回视频高
	获取视频帧率函数	返回视频帧率
	获取视频总帧数函数	返回视频总帧数
	获取视频封装格式函数	返回视频封装格式信息，类型为结构体
	释放资源函数	释放类所占的资源

6.2.3.2 视频编码任务类接口函数

视频编码任务类用于将视频算法处理后的视频帧数据进行编码。

初始化函数，目的是初始化视频编码任务类，函数参数应符合表18的规定。

表 18 视频编码任务类初始化函数参数列表

参数	输入/输出类型	参数说明
音视频编码类	输入参数	用于对视频编码任务类成员赋值

数据处理函数，是视频编码任务类的主体，派生自任务创建类，其实现了从编码一帧数据。

资源释放函数，目的是为了释放视频编码任务类所占资源，销毁类成员变量。

音视频编码类实现了视频的编码功能，其封装了编码视频和音频帧数据接口函数，支持配置输出视频文件的编码格式和像素格式等，类成员应符合表 19 的规定。

表 19 视频编码类主要成员说明表

名称	主要类成员列表	说明
音视频编码类	创建视频文件函数	根据传入的参数创建编码器和输出视频文件，具体参数见表20
	设置元数据信息函数	设置编码视频文件所需要的元数据信息，具体参数见6.2.2.5
	视频帧编码函数	编码一帧视频数据，并写入视频文件，输入为视频帧数据内存地址
	音频帧编码函数	对一帧音频包解码后重采样，再编码成音频包，并写入视频文件，输入为一帧音频数据包
	刷帧函数	用于刷出编码器中的帧，并写视频文件尾
	设置帧率函数	设置视频编码器帧率
	设置码率函数	设置视频编码器码率
	释放资源函数	释放类所占资源

表 20 创建视频文件函数参数信息

参数	输入/输出类型	参数说明
视频流编码格式类型	输入参数	见6.2.2.1
视频流像素格式	输入参数	见6.2.2.2
视频色域格式	输入参数	见6.2.2.3
HDR标准类型	输入参数	见6.2.2.4
音频流编码格式类型	输入参数	见6.2.2.6
视频文件封装类型	输入参数	见6.2.2.7
视频文件路径	输入参数	输出视频文件路径及文件名
视频文件宽	输入参数	输出视频帧的宽度
视频文件高	输入参数	输出视频帧的高度

6.2.3.3 音频编码任务类接口函数

音频编码任务类用于音频编码任务。

初始化函数，目的是初始化音频编码任务类，函数参数应符合表21的规定。

表 21 音频编码任务类初始化函数参数列表

参数	输入/输出类型	参数说明
音视频编码类	输入参数	用于对音频编码任务类成员赋值

数据处理函数，是音频编码任务类的主体，派生自任务创建类，其实现了从编码一帧音频数据。

资源释放函数，目的是为了释放音频编码任务类所占资源，销毁类成员变量。

6.2.3.4 GPU 分发任务类接口函数

GPU分发任务类按帧号顺序将解码后的视频帧数据分发给不同GPU上的算法任务。

初始化函数，目的是为了初始化 GPU 资源分发信息。 函数参数应符合表 22 的规定。

数据处理函数，目的是为了给每个 GPU 分发数据。

资源释放函数，目的是为了释放资源。

表 22 算法初始化函数参数列表

参数	输入/输出类型	参数说明
可用GPU资源信息容器	输入参数	容器中存放GPU资源的ID

6.2.3.5 GPU 集合任务类接口函数

GPU集合任务类用于把不同GPU上算法任务处理后视频帧数据按照帧号顺序进行集合。

初始化函数，目的是为了初始化 GPU 资源集合信息。 函数参数应符合表 23 的规定。

数据处理函数，目的是为了有序集合 GPU 数据。

资源释放函数，目的是为了释放资源。

表 23 算法初始化函数参数列表

参数	输入/输出类型	参数说明
可用GPU资源信息容器	输入参数	容器中存放GPU资源的ID

6.2.3.6 AI 模型处理任务类接口函数

AI模型处理任务类用于视频AI模型算法处理，包括但不限于超分，降噪，插帧，HDR。

初始化函数，目的是为了初始化 AI 模型任务，函数参数应符合表 24 的规定。

表 24 算法初始化函数参数列表

参数	输入/输出类型	参数说明
算法任务所需的AI模型类型	输入参数	见6.2.2.8

数据处理函数，目的是为了启动 AI 模型进行数据处理，并和其他算法任务进行数据传递。

资源释放函数，目的是释放任务资源。

6.2.4 算法任务接口配置流程

算法任务接口配置流程应符合图 5 的规定。

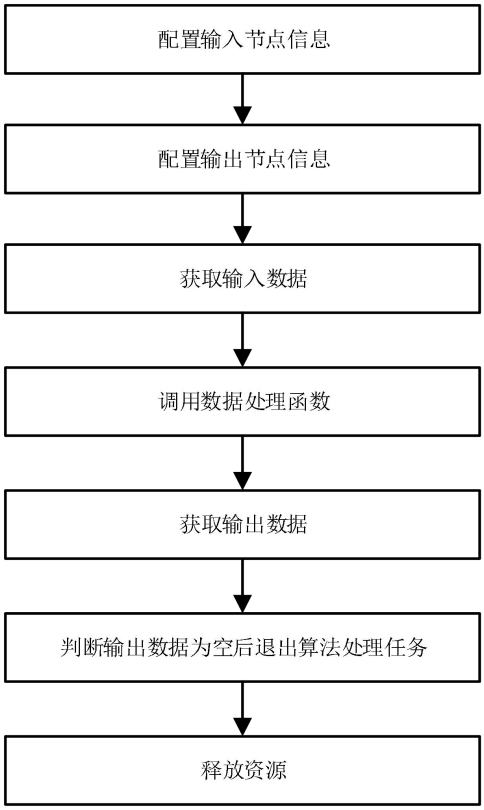

图 5 超高清算法任务接口配置流程

调用前向节点配置函数配置算法任务输入节点信息，调用后向节点配置函数配置算法任务输出节点信息，调用数据等待函数判断上一节点数据是否可读，并获取输入数据，调用数据处理函数对输入数据进行处理，调用传递数据函数判断下一节点数据是否可写，并给下一节点传递数据。若传入数据为空，则安全退出本次算法任务并释放资源。算法任务函数接口配合算法服务函数接口可实现多个算法任务的并行处理，并保障任务间的数据传递有序进行。

附 录 A
（资料性附录）
数据类型和接口函数原型

A.1 算法服务接口函数

A.1.1 数据同步类

```
class UnitSynArg{
    public:
        UnitSynArg();//构造函数
        int release();//资源释放函数
        sem_t readable_sem;//可读信号量
        sem_t writeable_sem;//可写信号量
        PassingPara *passing_para;//数据节点类
};
```

A.1.1.1 数据节点类

```
class PassingPara{
    public:
        std::map<const char *, void *, cmp_str> para_dict;//数据链表
        int init(PassingPara *passing_para);//初始化函数
        void *get_item(const char *);//获取数据函数
        int add_item(const char *, void *);//增加数据函数
        void *change_item(const char *, void *);//更改数据函数
        void *remove_item(const char *);//移除数据函数
};
```

A.1.1.1.1 初始化函数

函数原型：

```
int PassingPara::init(PassingPara *passing_para);
```

函数调用参数见表 A.1。

表 A.1 数据回调函数调用参数

参数	数据类型	输入/输出类型	参数说明
passing_para	PassingPara *	输入参数	要复制的数据节点对象

A.1.1.1.2 获取数据函数

函数原型：

void *PassingPara::get_item(const char *key);

函数调用参数见表 A.2。

表 A.2 数据回调函数调用参数

参数	数据类型	输入/输出类型	参数说明
key	const char *	输入参数	用于数据查找的关键字

A.1.1.1.3 增加数据函数

函数原型：

int PassingPara::add_item(const char *key, void *val);

增加数据函数调用参数见表 A.3。

表 A.3 数据回调函数调用参数

参数	数据类型	输入/输出类型	参数说明
key	const char *	输入参数	用于数据查找的关键字
val	void *	输入参数	内存数据的地址指针

A.1.1.1.4 更改数据函数

函数原型：

void *PassingPara::change_item(const char *key, void *val);

函数调用参数见表 A.4。

表 A.4 数据回调函数调用参数

参数	数据类型	输入/输出类型	参数说明
key	const char *	输入参数	用于数据查找的关键字
val	void *	输入参数	内存数据的地址指针

A.1.1.1.5 删除数据函数

函数原型：

void *PassingPara::remove_item(const char *key);

函数调用参数见表 A.5。

表 A.5　数据回调函数调用参数

参数	数据类型	输入/输出类型	参数说明
key	const char *	输入参数	用于数据查找的关键字

A.1.2　流程管理类

```
class VideoProcessFramework {
    public:
        int init();//初始化函数
        int add_task(FrameProcessUnit *frame_process_unit);//添加任务函数
        int start_tasks();//启动任务函数
        int close_tasks();//关闭任务函数
    private:
        static void* process_thread(void *arg);//算法任务线程函数
        std::vector<FrameProcessUnit *> process_list;//算法任务存储列表
};
```

A.1.2.1　添加任务函数

函数原型：

int VideoProcessFramework::add_task(FrameProcessUnit *frame_process_unit);

函数调用参数见表 A.6

表 A.6　添加任务函数调用参数

参数	数据类型	输入/输出类型	参数说明
frame_process_unit	FrameProcessUnit *	输入参数	指向算法任务单元的指针

A.1.3　任务创建类

```
class FrameProcessUnit{
```

```
    public:
        int pre_node(std::vector<UnitSynArg *> in_nodes);//前向节点配置函数
        int next_node(std::vector<UnitSynArg *> out_nodes);//后向节点配置函数
        int wait(int index, PassingPara **para_temp);//数据等待函数
        int post(int index, PassingPara *para_temp);//数据传递函数
        virtual int process(){};//数据处理函数
        virtual int close(){};//资源释放函数
    private:
        std::vector<UnitSynArg *> syn_in;//前向的同步节点
        std::vector<PassingPara *> para_in;//读取前向的数据传递
        std::vector<UnitSynArg *> syn_out; //后向的同步节点
        std::vector<PassingPara *> para_out;//读取后向的数据传递
};
```

A.1.3.1 前向节点配置函数

函数原型：

```
int FrameProcessUnit::pre_node(std::vector<UnitSynArg *> in_nodes);
```

函数调用参数见表 A.7

表 A.7 添加任务函数调用参数

参数	数据类型	输入/输出类型	参数说明
in_nodes	std::vector<UnitSynArg *>	输入参数	设置算法任务的前向节点

A.1.3.2 后向节点配置函数

函数原型：

```
int FrameProcessUnit::next_node(std::vector<UnitSynArg *> out_nodes);
```

函数调用参数见表A.8

表 A.8 后向节点配置函数调用参数

参数	数据类型	输入/输出类型	参数说明

in_nodes	std::vector<UnitSynArg *>	输入参数	设置算法任务的后向节点

A.1.3.3 数据等待函数

函数原型：

```
int FrameProcessUnit::wait(int index, PassingPara **para_temp);
```

函数调用参数见表 A.9

表 A.9 数据等待函数调用参数

参数	数据类型	输入/输出类型	参数说明
index	int	输入参数	算法任务前向数据节点的索引号
para_temp	PassingPara **	输入/输出参数	算法任务前向数据节点指针的指针

A.1.3.4 数据传递函数

函数原型：

```
int FrameProcessUnit::post(int index, PassingPara *para_temp);
```

函数调用参数见表 A.9

表 A.9 数据等待函数调用参数

参数	数据类型	输入/输出类型	参数说明
index	int	输入参数	算法任务后向数据节点的索引号
para_temp	PassingPara **	输入/输出参数	算法任务后向数据节点指针的指针

A.2 算法任务接口函数

A.2.1 音视频解码类

```
class VideoReader{
    public:
        int open_video(const char *filename);
        int read_frame(void *&data_out);
```

```
        int getwidth();
        int getheight();
        void get_fps(int &den, int &num);
        void get_nb_frames(int &nbframes);
        AVFormatContext *get_ifmt_ctx();
        int close();
    private:
        //视频的上下文信息
        AVFormatContext *ifmt_ctx = NULL;
        //解码帧
        AVFrame *frame = NULL;
        //媒体流
        AVStream *stream = NULL;
        //视频媒体流的编号
        int video_stream_index = -1;//添加视频流的index
        int audio_stream_index = -1;//添加音频流的index
        //编码上下文
        AVCodecContext *codec_ctx = NULL;
        //帧格式转换上下文
        SwsContext *swsctx = NULL;
        //帧转换输出解码帧
        AVFrame *out_frame = NULL;
        //输出的像素格式
        AVPixelFormat outpixfmt;
        //视频宽
        int width = 0;
        //视频高
        int height = 0;
```

```
        //输出图像的宽
        int output_width = 0;
        //输出图像的高
        int output_height = 0;
        //视频像素格式对应的每个像素字节数
        int pixbyte = 0;
    };
```

A.2.1.1 打开视频文件函数

函数原型：

```
int VideoReader::open_video(const char *filename);
```

函数调用参数见表A.10

表 A.10 打开视频文件函数调用参数

参数	数据类型	输入/输出类型	参数说明
filename	const char *	输入	视频文件路径

A.2.1.2 获取视频帧数据函数

函数原型：

```
int VideoReader::read_frame(void *&data_out)
```

获取视频帧数据函数调用参数见表A.11

表 A.11 获取视频帧数据函数调用参数

参数	数据类型	输入/输出类型	参数说明
data_out	void *&	输入/输出	视频/音频解码后的数据

A.2.2 音视频编码类

```
class VideoWriter{
    public:
        typedef struct metadataParam {
            char *key;
            metadata_ID value;
```

```
    } metadataParam;//元数据结构体
    //设置元数据信息函数
     int set_metadata_param(metadataParam* metadata_param);
    //创建视频文件函数
    int open_video(const char *filename,int width, int height,
                   Filefmt_ID file_id=0,
                   Codec_ID codec_id=0,
                   Pixfmt_ID pixfmt_id=0,
                   Hdr_ID hdr_id=0,
                   aCodec_ID acodec_id=0,
                   AVFormatContext *ifmt_ctx = NULL);
    int write_videoframe(uint8_t *frame);//视频帧编码函数
    int write_audiopacke(AVPacket *avpkt);//音频帧编码函数
    int flush();//刷帧函数
    int set_fps(int den, int num);//设置帧率函数
    int set_bitRate(int rate);//设置码率函数
    int close();//释放资源函数
private:
  std::vector<metadataParam *> metadata_param_list;//存储元数据信息的列表
    //输出文件上下文
    AVFormatContext* pFormatCtx = NULL;
    //输入文件上下文
    AVFormatContext *ifmt_ctx = NULL;
    //输出文件封装格式
    AVOutputFormat* fmt = NULL;
    //输出的视频流
    AVStream* video_st = NULL;
    AVStream* video_st_in = NULL;
```

```
AVStream* audio_st_in = NULL;

AVStream* audio_st_out = NULL;

int video_stream_index = -1;

int audio_stream_index = -1;

//编码上下文

AVCodecContext* pCodecCtx = NULL;

AVCodecContext* pAudioDeCodecCtx = NULL;

AVCodecContext* pAudioEnCodecCtx = NULL;

//编码器

AVCodec* pCodec = NULL;

AVCodec* pAudioDeCodec = NULL;

AVCodec* pAudioEnCodec = NULL;

//编码帧

AVPacket* packet = NULL;

//原始数据帧

AVFrame* pFrame = NULL;

//帧格式转换上下文

SwsContext *swsctx= NULL;

//编码器名字

const char *codec_name = NULL;

//编码ID

AVCodecID av_codec_id;

AVCodecID av_audio_codec_id;

//像素格式

AVPixelFormat config_pixfmt;

//视频的metadata

AVDictionary *dict = 0;

//写锁
```

```
pthread_mutex_t write_lock;
//帧数统计
long int frame_cnt = 0;
//视频的宽
int width = 0;
//视频的高
int height = 0;
//帧率
int fps_den = 1;
int fps_num = 25;
//码率
long int bit_rate = 100000000;
//hdr模式开关
int hdr_on = 0;
//创建输出视频文件函数
int open_output_file(const char *filename, int widht, int height);
//音频重采样编码相关ffmpeg函数
SwrContext *audio_resampler_context = NULL;
AVAudioFifo *audio_fifo = NULL;
int init_fifo(AVAudioFifo **fifo, AVCodecContext *output_codec_context);
int convert_samples(const uint8_t **input_data,
                    uint8_t **converted_data, const int frame_size,
                    SwrContext *resample_context);
int init_resampler(AVCodecContext *input_codec_context,
                   AVCodecContext *output_codec_context,
                   SwrContext **resample_context);
init_converted_samples(uint8_t ***converted_input_samples,
                              AVCodecContext *output_codec_context,
```

```
                                int frame_size);
        add_samples_to_fifo(AVAudioFifo *fifo,
                                uint8_t **converted_input_samples,
                                const int frame_size);
};
```

A.2.2.1 创建视频文件函数

函数原型：

```
int VideoWriter::open_video(const char *filename,int width, int height,
                Filefmt_ID file_id=0,
                Codec_ID codec_id=0,
                Pixfmt_ID pixfmt_id=0,
                Hdr_ID hdr_id=0,
                aCodec_ID acodec_id=0,
                AVFormatContext *ifmt_ctx = NULL);
```

函数调用参数见表A.12

表 A.12 创建视频文件函数调用参数

参数	数据类型	输入/输出类型	参数说明
filename	const char *	输入	输出视频文件的路径和文件名
width	int	输入	输出视频文件的宽度
height	int	输入	输出视频文件的高度
file_id	Filefmt_ID	输入	输出视频文件的封装格式，默认为 mp4
codec_id	Codec_ID	输入	视频编码器的类型，默认为视频 H264
pixfmt	Pixfmt_ID	输入	视频编码帧的像素格式，默认为 YUV420P
hdr_id	Hdr_ID	输入	视频文件 HDR 标准类型，默认为无 HDR
acodec_id	aCodec_ID	输入	音频编码器的类型，默认为 AAC

ifmt_ctx	AVFormatContext *	输入	源视频文件的封装环境，可以选择性的为输出视频文件对应的封装环境赋值

A.2.2.2 视频帧编码函数

函数原型：int VideoWriter::write_videoframe(uint8_t *frame);

函数调用参数见表A.13

表 A.13 视频帧编码函数调用参数

参数	数据类型	输入/输出类型	参数说明
frame	uint8_t *	输入	视频帧数据指针

A.2.2.3 音频帧编码函数

函数原型：int VideoWriter::write_audiopacket(AVPacket *avpkt);

函数调用参数见表A.14

表 A.14 音频帧编码函数调用参数

参数	数据类型	输入/输出类型	参数说明
avpkt	AVPacket *	输入	音频数据包

A.2.2.4 设置帧率函数

函数原型：

```
int VideoWriter::set_fps(int den, int num)
```

函数调用参数见表A.15

表 A.15 设置帧率函数调用参数

参数	数据类型	输入/输出类型	参数说明
den	int	输入	时间基的分子
num	int	输入	时间基的分母

A.2.2.5 设置码率函数

函数原型：

```
int VideoWriter::set_bitRate(long int bit_rate);
```

函数调用参数见表A.16

表 A.16　设置码率函数调用参数

参数	数据类型	输入/输出类型	参数说明
bit_rate	long int	输入	输出视频文件的码率

A.2.2.6　设置元数据信息函数

函数原型：

```
int set_metadata_param(metadataParam* metadata_param);
```

函数调用参数见表A.17

表 A.17　设置元数据信息函数调用参数

参数	数据类型	输入/输出类型	参数说明
metadata_param	metadataParam*	输入	指向元数据信息结构体的指针

A.2.3　音视频解码任务类

```
class VideoReaderUnit : public FrameProcessUnit{
    public:
        int init(const char *input_path);
        AVFormatContext *get_ifmt_ctx();
        int process() override;
        int close() override;
        int get_fps(int &den, int &num);
        int get_nb_frames(int &nbframes);
        BOEmm::VideoReader video_reader;
};
```

A.2.3.1　初始化函数

函数原型：

```
int VideoReaderUnit::init(const char *input_path) ;
```

函数调用参数见表A.18

表 A.18　初始化函数调用参数

参数	数据类型	输入/输出类型	参数说明
input_path	const char *	输入	视频文件的地址和文件名

A.2.3.2　获取视频帧率函数

函数原型：

```
int VideoReaderUnit::get_fps(int &den, int &num);
```

函数调用参数见表A.19

表 A.19　获取视频帧率函数调用参数

参数	数据类型	输入/输出类型	参数说明
den	int	输入/输出	时间基的分子
num	int	输入/输出	时间基的分母

A.2.3.3　获取视频总帧数函数

函数原型：

```
int VideoReaderUnit::get_nb_frames(int &nbframes);
```

函数调用参数见表A.20

表 A.20　获取视频总帧数函数调用参数

参数	数据类型	输入/输出类型	参数说明
nbframes	int &	输入/输出	视频文件的总帧数

A.2.4　视频编码任务类

```
class VideoWriterImgUnit : public FrameProcessUnit{
        int init(BOEmm::VideoWriter *video_writer);//初始化函数
        int process() override;//数据处理函数
        int close() override;//资源释放函数
    private:
        BOEmm::VideoWriter *video_writer;//音视频编码类
};
```

A.2.4.1 初始化函数

函数原型：

int VideoWriterImgUnit::init(BOEmm::VideoWriter *video_writer）;

函数调用参数见表A.21

表 A.21 初始化函数调用参数

参数	数据类型	输入/输出类型	参数说明
video_writer	BOEmm::VideoWriter *	输入	音视频编码类指针

A.2.5 音频编码任务类

```
class VideoWriterPktUnit : public FrameProcessUnit{
    public:
        int init(BOEmm::VideoWriter *video_writer);//初始化函数
        int process() override;//数据处理函数
        int close() override;//资源释放函数
    private:
        BOEmm::VideoWriter *video_writer;//音视频编码类
};
```

A.2.5.1 初始化函数

函数原型：

int VideoWriterPktUnit::init(BOEmm::VideoWriter *video_writer);

函数调用参数见表A.22

表 A.22 初始化函数调用参数

参数	数据类型	输入/输出类型	参数说明
video_writer	BOEmm::VideoWriter *	输入	音视频编码类指针

A.2.6 GPU分发任务类

```
class OneToManyUnit : public FrameProcessUnit{
```

```
public:
    int init(vector<int> vecdeviceid);//初始化函数
    int process() override;//数据处理函数
    int close() override;//资源释放函数
private:
    int numGPU = 0;//总GPU数
    int cnt = 0;//计数
};
```

A.2.6.1 初始化函数

函数原型：

```
int OneToManyUnit::init(vector<int> vecDeviceID) ;
```

函数调用参数见表A.23

表 A.23 初始化函数调用参数

参数	数据类型	输入/输出类型	参数说明
vecDeviceID	vector<int>	输入	容器中存放 GPU 资源的 ID

A.2.7 GPU集合任务类

```
class ManyToOneUnit : public FrameProcessUnit{
public:
    int init(vector<int> vecdeviceid);//初始化函数
    int process() override;//数据处理函数
    int close() override;//资源释放函数
private:
    int numGPU = 0;//总GPU数
    int cnt = 0;//计数
};
```

A.2.7.1 初始化函数

函数原型：

```
init(vector<int> vecdeviceid);
```

函数调用参数见表A.24

表 A.24　初始化函数调用参数

参数	数据类型	输入/输出类型	参数说明
vecDeviceID	vector<int>	输入	容器中存放 GPU 资源的 ID

A.2.8　AI模型处理任务类

```
class modelProcessUnit : public FrameProcessUnit{
public:
    int init(BOEmm::eumMODEL_ID modelID, int gpu_id);//初始化函数
    int process() override;//数据处理函数
    int close() override;//资源释放函数
private:
    int getModelConfig(BOEmm::eumMODEL_ID modelID,modelConfigInfo& thisConfig);//获取模型配置信息函数
    myconfigInfo model_config[MODEL_NUM];//模型配置信息表
    trtfilter *model = NULL;//模型tensorrt推理引擎指针
};
```

A.2.8.1　初始化函数

函数原型：

```
int modelProcessUnit::init(BOEmm::eumMODEL_ID modelID ,int gpu_id) ;
```

函数调用参数见表A.25

表 A.25　初始化函数调用参数

参数	数据类型	输入/输出类型	参数说明
modelID	BOEmm::eumMODEL_ID	输入	模型枚举号
gpu_num	int	输入	GPU 的 id 号

A.3　调用示例

```
int main(int argc, char *argv[]){
    int gpu_num =0;//GPU 数量
    cudaSts = cudaGetDeviceCount(&gpu_num);
    cudaGetDeviceCount(&gpu_num);
    char* pInputVideo = "./test_video/2022-08-09-172550.mov";//输入视频路径和文件名
    char* pOutVideo = "./out_video/";//输出视频路径
    long int nBitRate = 88888888;//输出视频码率
    //输出视频编码信息格式
    BOEmm::Codec_ID codec_id = BOEmm::H264;
    BOEmm::Pixfmt_ID pixfmt_id = BOEmm::YUV420;
        vector<BOEmm::eumMODEL_ID> vecTrtModelID;//AI 算法模型容器
        vecTrtModelID.push_back(BOEmm::SR_1080);//添加超分模型
        vecTrtModelID.push_back(BOEmm::HDR_2160);//添加 HDR 模型
        int nw = 1920;//输入视频解码目标宽度
        int nh = 1080;//输入视频解码目标高度
        int nc = 3;//输入视频通道数
        int nscale = 2;//超分模型拉伸倍率
        int ishdr = 1;//是否是 hdr 模式
        gpu_num = vecdeviceid.size();//gpu 数量
    //初始化
    int nw_sr = nw*nscale;//输出视频宽
    int nh_sr = nh*nscale;//输出视频高
    VideoProcessFramework video_process_framework;//创建视频流水线类
    BOEmm::VideoReader*video_reader = new BOEmm::VideoReader(pInputVideo, nw, nh, true);//创建音视频解码类
    VideoReaderUnit video_reader_unit;//创建视频解码任务类
    sdkSts = video_reader_unit.init(video_reader);
    BOEmm::VideoWriter *video_writer = new BOEmm::VideoWriter();//创建音视频编码类
    int den = 0, num = 0;
```

```
int nbframes;
video_reader_unit.get_fps(den, num);
video_reader_unit.get_nb_frames(nbframes);
video_writer->set_fps(den, num);
video_writer->set_bitRate(nBitRate);
sdkSts = video_writer->open_video(pOutVideo,nw_sr, nh_sr,
codec_id, pixfmt_id, video_reader_unit.get_ifmt_ctx());
VideoWriterImgUnit video_writer_img_unit;//创建视频编码任务
video_writer_img_unit.init(video_writer,iscopyfrc,frc,ishdr);
std::vector<UnitSynArg *> node_videoWriteUnit_in;
VideoWriterPktUnit video_writer_pkt_unit;//创建音频编码任务
video_writer_pkt_unit.init(video_writer);
//视频文件解码后输出视频和音频数据，所以有两个同步节点
//将两个同步节点打包
UnitSynArg *synArg_videoReadUint_img_out = new UnitSynArg();//视频数据节点
UnitSynArg *synArg_videoReadUint_pkt_out = new UnitSynArg();//音频数据节点
std::vector<UnitSynArg *> videoReadUint_node_out = {synArg_videoReadUint_img_out,
synArg_videoReadUint_pkt_out};
video_reader_unit.next_node(videoReadUint_node_out);//配置视频解码任务后向节点
FrameProcessUnit *video_reader_process_unit = &video_reader_unit;//
video_process_framework.add_task(video_reader_process_unit);//将视频解码任务添加到视频处理流水线中
std::vector<UnitSynArg *> node_videoPktwWriter_in = {synArg_videoReadUint_pkt_out};
//配置写音频的入口
video_writer_pkt_unit.pre_node(node_videoPktwWriter_in);
//将音频编码任务加入到视频处理流水线
FrameProcessUnit *video_writer_pkt_process_unit = &video_writer_pkt_unit;
video_process_framework.add_task(video_writer_pkt_process_unit);
int model_nums = vecTrtModelID.size();
```

```
//分发器
OneToManyUnit one_to_many;
one_to_many.init(vecdeviceid);
std::vector<UnitSynArg *> node_oneToManyUint_in = {synArg_videoReadUint_img_out}; // 分发器的入口为video_reader 类的视频数据出口
one_to_many.pre_node(node_oneToManyUint_in);
std::vector<UnitSynArg *> node_oneToManyUint_out;//
//many_to_one 集成器
ManyToOneUnit many_to_one;
many_to_one.init(vecdeviceid);
std::vector<UnitSynArg *> node_ManyToOneUint_in;//
std::vector<UnitSynArg *> node_ManyToOneUint_out;//
//根据 GPU 数量对 AI 算法模型任务进行初始化
TensorrtProcessUnit tensorrt_model[gpu_num];
for(int i = 0; i < gpu_num; i++)
{
    tensorrt_model[i].init(vecTrtModelID, vecdeviceid,i,ishdr);
}
//循环每个模型处理模型
for(int i = 0; i < gpu_num; i++)
{
    //每个模型一个入口一个出口节点
    UnitSynArg *synArg_trtModelUnit_in = new UnitSynArg();//
    UnitSynArg *synArg_trtModelUnit_out = new UnitSynArg();
    node_oneToManyUint_out.push_back(synArg_trtModelUnit_in);//分发器的出口为多个 gpu,trt 模型单元的入口
    node_ManyToOneUint_in.push_back(synArg_trtModelUnit_out);//集合器的入口为多个 gpu,trt 模型单元的出口
    //配置 AI 算法模型处理任务的入口节点
    std::vector<UnitSynArg *> node_trtModelUint_in = {synArg_trtModelUnit_in};
```

```
        tensorrt_model[i].pre_node(node_trtModelUint_in);
        //配置 AI 算法模型处理任务的出口节点
        std::vector<UnitSynArg *> node_trtModelUint_out = {synArg_trtModelUnit_out};
        tensorrt_model[i].next_node(node_trtModelUint_out);
        //将 AI 算法模型处理任务加入视频处理流水线中
        FrameProcessUnit *tensorrt_model_process_unit = &(tensorrt_model[i]);
    video_process_framework.add_task(tensorrt_model_process_unit);
}
 //将分发任务加入视频处理流水线中
 one_to_many.next_node(node_oneToManyUint_out);
 FrameProcessUnit *one2many_unit = &one_to_many;
 video_process_framework.add_task(one2many_unit);
 //AI 模型任务的出口即为 many_to_one 集成器的入口
 many_to_one.pre_node(node_ManyToOneUint_in);
 //配置模型出口节点
 UnitSynArg * synArg_mangToOneUint_Out = new UnitSynArg;
 syn_list.push_back(synArg_mangToOneUint_Out);
 node_ManyToOneUint_out.push_back(synArg_mangToOneUint_Out);
 many_to_one.next_node(node_ManyToOneUint_out);
 //模型集合器加入视频处理流水线中
 FrameProcessUnit *many2one_unit = &many_to_one;
 video_process_framework.add_task(many2one_unit);
 node_videoWriteUnit_in.push_back(synArg_mangToOneUint_Out);
//配置视频编码任务的入口节点
video_writer_img_unit.pre_node(node_videoWriteUnit_in);
FrameProcessUnit *video_writer_img_process_unit = &video_writer_img_unit;
//视频编码任务加入视频处理流水线中
video_process_framework.add_task(video_writer_img_process_unit);
```

```
    //启动任务，等到所有任务结束返回。
    video_process_framework.start_tasks();
    video_process_framework.close_tasks();
}
```

附 录 B
（资料性附录）
函数返回错误码详细说明

算法服务和算法任务接口函数返回错误码以枚举形式列出，详细说明如下。

```
enum mmsdkErrorSts
{
//AVfile
    mmsdkErrorAVOpenInputFileFailed                = -1000,
    mmsdkErrorAVOpenOutputFileFailed,
    mmsdkErrorAVInputFileReadFinshed,
    mmsdkErrorAVerrorEOF,
    mmsdkErrorAVFileHeaderWriteFailed,
    mmsdkErrorAVFileTrailerWriteFailed,

    //reader
    mmsdkErrorAVVideofillArrayFailed,
    //fmttxt
    mmsdkErrorAVfmtWrong,
    //decoderTxt.encoderTxt
    mmsdkErrorAVAudioCodecTxtParameterCopyFailed,
    mmsdkErrorAVAudioCodecTxtAllocateFailed,
    mmsdkErrorAVVideoCodecTxtParameterCopyFailed,
    mmsdkErrorAVVideoCodecTxtAllocateFailed,
    //decoder
    mmsdkErrorAVAudioDecoderOpenFailed,
    mmsdkErrorAVAudioDecoderFindFailed,
    mmsdkErrorAVVideoDecoderOpenFailed,
    mmsdkErrorAVVideoDecoderFindFailed,
```

```
        //encoder
        mmsdkErrorAVAudioEncoderFindFailed,
        mmsdkErrorAVAudioEncoderOpenFailed,
        mmsdkErrorAVVideoEncoderFindFailed,
        mmsdkErrorAVVideoEncoderOpenFailed,
        //stream
        mmsdkErrorAVAudioStreamCreateFailed,
        mmsdkErrorAVAudioStreamInfoFailed,
        mmsdkErrorAVAudioStreamAllocateFailed,
        mmsdkErrorAVVideoStreamCreateFailed,
        mmsdkErrorAVVideoStreamInfoFailed,
        mmsdkErrorAVVideoStreamAllocateFailed,
        //swr
        mmsdkErrorAVAudioSwrCtxCallocFailed,
        mmsdkErrorAVAudioSwrCtxAllocFailed,
        mmsdkErrorAVAudioSwrCtxInitFailed,
        mmsdkErrorAVAudioSamplesSwrConvertFailed,
        mmsdkErrorAVAudioSwrConvertedSamplesPointerCallocFailed,
        mmsdkErrorAVAudioSwrConvertedSamplesAllocateFailed,
        mmsdkErrorAVVideoSwscaleFailed,
        //packet
        mmsdkErrorAVAudioPacketReadFailed,
        mmsdkErrorAVAudioPacketSendFailed,
        mmsdkErrorAVVideoPacketReadFailed,
        mmsdkErrorAVVideoPacketReceiveFailed,
        mmsdkErrorAVAudioPacketReceiveFailed,
        mmsdkErrorAVVideoPacketSendFailed,
        mmsdkErrorAVAudioAVPacketRescaleTsFailed,
```

```
    mmsdkErrorAVVideoAVPacketRescaleTsFailed,
    mmsdkErrorAVVideoPacketInterLeavedWriteFailed,
    mmsdkErrorAVAudioPacketInterLeavedWriteFailed,
    //frame
    mmsdkErrorAVAudioReceiveFrameFailed,
    mmsdkErrorAVVideoReceiveFrameFailed,
    mmsdkErrorAVAudioFrameSendFailed,
    mmsdkErrorAVVideoFrameSendFailed,
    mmsdkErrorVideoFrameSwsScaleFailed,
    mmsdkErrorAVAudioFrameAllocateFailed,
    mmsdkErrorAVVideoFrameAllocateFailed,
    mmsdkErrorAVAudioFrameGetBufferFailed,
    mmsdkErrorAVVideoPtsWorng,
    //audiofifo
    mmsdkErrorAVFifoAllocateFailed,
    mmsdkErrorAVFifoReAllocateFailed,
    mmsdkErrorAVFifoReadFailed,
    mmsdkErrorAVFifoWriteFailed,
    //flush
    mmsdkErrorAVFrameFlushFailed,
    mmsdkErrorAVPacketFlushFailed,
//CUDA
    mmsdkErrorCUDADeviceFailed              = -800,
    mmsdkErrorCUDAMallocFailed,
    mmsdkErrorCUDAMemcpyDeviceToHostFailed,
    mmsdkErrorCUDAMemcpyHostToDeviceFailed,
    mmsdkErrorCUDAMemcpyDeviceToDeviceFailed,
//Filter Unit
```

```
        mmsdkErrorUnitWaitWorng                 = -700,
        mmsdkErrorUnitPostWorng,
        mmsdkErrorParaAdditemWorng,
        mmsdkErrorUnitVideoReadExit,
        mmsdkErrorUnitImgWriterExit,
        mmsdkErrorUnitAudioWriterExit,
        mmsdkErrorUnitTrtModelExit,
        mmsdkErrorUnitOneToMannyExit,
        mmsdkErrorUnitMannyToOneExit,
        mmsdkErrorUnitHostToDeviceExit,
        mmsdkErrorUnitDeviceToHostExit,
        mmsdkSuccess = 0
    };
```

附 录 C
（资料性附录）
依赖库版本详细说明

C.1 依赖库版本说明

表C.1列举的版本为推荐版本，不代表其他版本不适用本标准。

使用第三方依赖库版本详细说明见表C.1

表 C.1 第三方依赖库版本说明表

依赖库名称	版本号
ffmpeg	4.3.2
cuda	11.2
cudnn	8.0